Objective

- The main goal of this course is to present techniques for the design and analysis of algorithms that are "efficient" with respect to the use of some resource(s), such as time, space, processors, or the number of operations of a particular type (e.g., additions, comparisons)
- Of course, we also require our algorithms to be correct!

Algorithmic Efficiency

- A computational problems that we address typically corresponds to an infinite family of instances
  - For example, consider the problem of comparison-based sorting of n distinct keys
  - There are infinitely many possible values of n, and for each value of n, there are effectively $n!$ different instances
  - For a given (correct) algorithm $\mathcal{A}$, let $T_{\mathcal{A}}(n)$ denote the worst-case number of comparisons performed by $\mathcal{A}$ over all $n!$ instances with n keys
  - It is natural to seek an algorithm $\mathcal{A}$ minimizing $T_{\mathcal{A}}(n)$
  - For most of the problems that we consider, it is difficult/hopeless to achieve absolute optimality; instead, we seek algorithms that are asymptotically efficient

Asymptotic Notation

- Suppose we want to compare the performance of two comparison-based sorting algorithms $\mathcal{A}$ and $\mathcal{B}$
  - Assume that we have established upper bounds of $f(n)$ and $g(n)$, respectively, on the worst-case number of comparisons performed by algorithms $\mathcal{A}$ and $\mathcal{B}$
  - Which is the better upper bound?
  - In asymptotic analysis, we focus on comparing $f(n)$ and $g(n)$ as n tends to ∞
  - For the kind of (asymptotically nonnegative) functions $f(n)$ and $g(n)$ that we typically encounter in the analysis of algorithms, $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is either 0, some positive constant (i.e., independent of the parameter n), or ∞

Big Oh

- We write $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$
- Remark: If we do not restrict the functions $f(n)$ and $g(n)$ to be asymptotically nonnegative, then we typically define $f(n)$ to be $O(g(n))$ if there exist positive constants n_0 and c such that $|f(n)| \leq c \cdot |g(n)|$ for all $n \geq n_0$
- For the kind of (asymptotically nonnegative) functions $f(n)$ and $g(n)$ that we typically encounter in the analysis of algorithms, we have $f(n) = O(g(n))$ when $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
- As such, $f(n) = O(g(n))$ asserts that $g(n)$ is a "fuzzy" (up to a constant factor) upper bound on $g(n)$

Big Omega

- How do we express that $g(n)$ is a "fuzzy" lower bound on $g(n)$?
- We write $f(n) = \Omega(g(n))$
- Just as $a \leq b$ if and only if $b \geq a$ for any real numbers a and b, we have $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

The Θ Symbol

- How do we express that $g(n)$ is a "fuzzy" tight bound on $g(n)$?
- We write $f(n) = \Theta(g(n))$
- Just as $a = b$ if and only if $a \leq b \ \wedge \ a \geq b$ for any real numbers a and b, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Little-oh, Little-omega

- How do we express that $f(n)$ grows asymptotically more slowly than $g(n)$ (i.e., $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$)?
- We write $f(n) = o(g(n))$
- Likewise, we write $f(n) = \omega(g(n))$ to express that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

The $\sim$ Symbol

- Sometimes we wish to express that $f(n)$ is equal to $g(n)$ up to lower-order terms
- This may be written as $f(n) \sim g(n)$, which is pronounced "$f(n)$ is asymptotic to $g(n)$"
- For example, $12n^2 + 7n - 22 \sim 12n^2$
- Such a relationship is sometimes expressed using the o symbol as follows: $f(n) = (1 + o(1))g(n)$
  - While the first-order terms that we encounter in the design and analysis of algorithms are typically nonnegative, second-order terms can easily be negative
  - In work focusing on second-order terms, it is sometimes more convenient to use the definitions of the symbols O, Ω, Θ, o, ω based on absolute value

The Door-in-the-Wall Puzzle

You are facing a wall that extends infinitely in both directions. There is a small gap in the wall some positive integer number n steps away, but you don't know how far or in which direction. (Assume the gap is small and you can only detect it when standing directly in front of it.)

Here is an algorithm that guarantees finding the gap within a finite number of steps: Explore 1 step away in each direction, then 2 away, then 3 away, et cetera.

What is a Θ-bound on the worst-case number of steps taken by this algorithm?

A Better Algorithm for the Door-in-the-Wall Puzzle

In our initial solution, the sequence of interval sizes explored forms an arithmetic sequence. Instead, it is better to induce a geometric sequence by increasing the size of the search interval by a multiplicative factor (e.g., a factor of 2) at each iteration.

What is a Θ-bound on the worst-case number of steps taken by the revised algorithm?

A Family of Divide-and-Conquer Recurrences

- Consider the class of recurrences given by $T(1) = O(1)$ and

$$T(n) \leq aT(n/b) + n^c$$

  where n, a, and b are positive integers, n is a positive integer power of b, and $c \geq 0$
- The term n^c above is sometimes referred to as the "overhead term" of the recurrence
- We wish to obtain a (best-possible) O-bound for $T(n)$
- One nice way to do this is to analyze the recursion tree

Analysis of the Recursion Tree

- The total overhead associated with the root is n^c
- The total overhead associated with the a children of the root is $a(n/b)^c = (a/b^c)n^c$
- The total overhead associated with the a^2 grandchildren of the root is $a^2(n/b^2)^c = (a/b^c)^2 n^c$
- Detecting a pattern, we see that the total overhead associated with the a^i nodes at depth i, $0 \leq i < \log_b n$, is $(a/b^c)^i n^c$
- This is a geometric sequence with ratio a/b^c
- We also need to account for the cost of evaluating the leaves

The Cost of Evaluating the Leaves

- The depth of the recursion tree is $\log_b n$
- The number of nodes grows by a factor of a at each level
- Thus the number of leaves is $a^{\log_b n}$
- Observe that $a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}$
- Thus the number of leaves is $n^{\log_b a}$, and the total cost of evaluating the leaves is $O(n^{\log_b a})$

Case 1: $a = b^c$

- The total overhead at each level is the same (n^c)
- Hence $T(n) \leq O(n^{\log_b a}) + n^c(\log_b n) = O(n^c \log n)$
  - Notice that $a = b^c$ implies $\log_b a = c$
- Example: Mergesort, which has $a = b = 2$ and $c = 1$
- Remark: Changing the overhead term to dn^c for some positive constant d does not affect the asymptotic complexity of $T(n)$, since it merely scales all of the overhead terms by d

Case 2: $a < b^c$

- The total overhead is dominated by the overhead at the root, which is n^c
- The total cost of processing the leaves is $O(n^{\log_b a}) = o(n^c)$
- Thus $T(n) = O(n^c)$
- Example: Some linear-time algorithms are based on this recurrence with $a = 1$, $b = 2$, and $c = 1$

Case 3: $a > b^c$

- The total overhead is dominated by the term associated with depth $\log_b n - 1$, i.e., the parents of the leaves
- Each such node has constant size (namely b) and hence constant overhead (b^c), and the number of such nodes is a constant fraction ($1/a$) of the number of leaves
- It follows that $T(n)$ is proportional to the number of leaves, i.e., $T(n) = O(n^{\log_b a})$
- Example: Strassen's matrix multiplication algorithm has $a = 7$ and $b = c = 2$, and hence $T(n) = O(n^{\log_2 7})$

Some Linear-Time Recurrences

- $T(O(1)) = O(1)$ and $T(n) \leq T(\lceil \alpha n \rceil) + O(n)$ for some positive constant $\alpha < 1$
- Using the recursion tree method (or other methods, such as induction), it is easy to argue that $T(n) = O(n)$
- The following generalization likewise yields $T(n) = O(n)$: $T(O(1)) = O(1)$ and $T(n) \leq \sum_{1 \leq i \leq k} T(\lceil \alpha_i n \rceil) + O(n)$ where the α_i's are positive constants summing to less than 1
- A famous example: The Blum-Floyd-Pratt-Rivest-Tarjan (BFPRT) linear-time selection algorithm

The (Comparison-Based) Selection Problem

- The input is an unsorted collection of n "keys", and a desired rank k, $1 \leq k \leq n$
  - For simplicity, let us assume that the keys are distinct
  - The minimum key has rank 1, the next smallest key has rank 2, et cetera
- The output is the key of rank k
- Keys can only be compared

Sketch of the BFPRT Algorithm

- In this sketch, we will ignore a few minor details (e.g., in the next step, we assume for simplicity that n is a multiple of 101)
- Choose a sufficiently large odd number, say 101, and partition the n keys into $n/101$ groups of size 101
- Compute the $n/101$ group medians in $O(n)$ time
- Recursively compute the median of the medians $n/101$ group medians; let's call this x

Sketch of the BFPRT Algorithm (cont'd)

- Partition the n keys into the set of keys A that are less than x, the singleton set $\{x\}$, and the set of remaining keys B
- If the desired rank r is equal to $|A| + 1$, halt and return x
- If the desired rank r is at most $|A|$, recursively select the key with rank r in A
- Otherwise, recursively select the key with rank $r - |A| - 1$ in B

Analysis of the BFPRT Algorithm

- The minimum possible rank of x is approximately $51(n/101)(1/2) \approx n/4$
- It is not hard to argue that (for n sufficiently large) the rank of x is at least $n/5$
- Likewise, the rank of x is at most $4n/5$
- Thus we obtain the recurrence $T(O(1)) = O(1)$ and

$$T(n) \leq T(n/101) + T(4n/5) + O(n),$$

which solves to give $T(n) = O(n)$ since $\frac{1}{101} + \frac{4}{5} < 1$

More Divide-and-Conquer Recurrences

- Many variants of the D&C family considered earlier can be solved using the same methodology
- Assuming $T(O(1)) = O(1)$, and ignoring minor details related to integrality et cetera, solve each of the following recurrences
  - $T(n) \leq 2T(\sqrt{n}) + \log_2 n$
  - $T(n) \leq \sqrt{n}T(\sqrt{n}) + n$
  - $T(n) \leq 2T(n/2) + n \log n$
  - $T(n) \leq 2T(n/2) + n(\log n)^k$ where k is a nonnegative integer
  - $T(n) \leq 2T(n/2) + \frac{n}{\log_2 n}$

A Puzzle

- Specify a function $f(n)$ for which the recurrence $T(n) \leq (n/f(n))T(f(n)) + n$ (and $T(O(1)) = O(1)$) yields $T(n) = O(n \log^{(3)} n)$
  - Previously we have seen that if $f(n) = n/2$ then $T(n) = O(n \log n)$, and if $f(n) = \sqrt{n}$ then $T(n) = O(n \log \log n) = O(n \log^{(2)} n)$
- What if 3 is replaced by an integer $i > 3$?

A Lower Bound for Comparison-Based Sorting

- Recall that Mergesort uses $O(n \log n)$ comparisons
- We claim that any deterministic comparison-based sorting algorithm uses $\Omega(n \log n)$ comparisons
- Such an algorithm needs to determine which of the $n!$ possible permutations corresponds to the given input
- Let S_i denote the set of permutations that are consistent with the outcomes of the first i comparisons made by the algorithm
- If $|S_i| > 1$, the algorithm cannot terminate
- No matter what comparison the algorithm makes, there is a possible outcome for the comparison such that $|S_{i+1}| \geq |S_i|/2$

A Lower Bound for Comparison-Based Sorting (cont'd)

- It follows that the worst-case number of comparisons is at least $\log_2 n!$
- Using the crude inequality $n! \geq (n/2)^{n/2}$, we obtain the desired $\Omega(n \log n)$ lower bound
- In fact, this is a case where we can obtain upper and lower bounds that are tight to within lower-order terms
  - The worst-case number of comparisons needed to merge two sorted lists of length $n/2$ is at most n, so Mergesort uses at most $n \log_2 n$ comparisons (assume for simplicity that n is a power of 2)
  - Stirling's approximation gives $n! = \sqrt{2\pi n}(n/e)^n(1 + O(1/n))$, and hence $\log_2 n! = n \log_2 n - (\log_2 e)n + O(\log n) \sim n \log_2 n$

An $O(n \log n)$-Comparison "Top-Down" Sorting Algorithm

- Mergesort works "bottom-up" since it only does a "global calculation" at the end
- We can obtain an $O(n \log n)$-comparison top-down sorting algorithm by using the BFPRT algorithm to compute the median x in $O(n)$ time, partitioning about x, and recursively sorting the set of at most $n/2$ keys less than x and the set of at most $n/2$ keys greater than x
- The resulting recurrence is the same as for Mergesort, except that the overhead term (and hence also the final bound) is increased by a constant factor

Another Sorting-Related Question

- If we "preprocess" an unsorted set of n distinct keys by sorting it, we can search for a given key using $O(\log n)$ comparisons by binary search
- Thus, with $O(n \log n)$ preprocessing complexity, we can achieve $O(\log n)$ search compexity
- Show that with $O(n \log \log n)$ preprocessing complexity, we can achieve $O(n/\log^c n)$ search complexity for any positive constant c
  - Sometimes a result like this is stated more tersely as follows: With $O(n \log \log n)$ preprocessing complexity, we can achieve $O(n/\operatorname{polylog}(n))$ search complexity